Application 2 : Real-Time Data Processing System for Weather Monitoring with Rollups and Aggregates

Objective:

Develop a real-time data processing system to monitor weather conditions and provide summarized insights using rollups and aggregates. The system will utilize data from the OpenWeatherMap API (https://openweathermap.org/).

Data Source:

The system will continuously retrieve weather data from the OpenWeatherMap API. You will need to sign up for a free API key to access the data. The API provides various weather parameters, and for this assignment, we will focus on:

● main: Main weather condition (e.g., Rain, Snow, Clear)

● temp: Current temperature in Centigrade

● feels_like: Perceived temperature in Centigrade

● dt: Time of the data update (Unix timestamp)

Processing and Analysis:

● The system should continuously call the OpenWeatherMap API at a configurable interval (e.g., every 5 minutes) to retrieve real-time weather data for the metros in India. (Delhi, Mumbai, Chennai, Bangalore, Kolkata, Hyderabad)

● For each received weather update: ○ Convert temperature values from Kelvin to Celsius (tip : you can also use user preference).

Rollups and Aggregates:

1. Daily Weather Summary:

○ Roll up the weather data for each day.
○ Calculate daily aggregates for:
  ■ Average temperature
  ■ Maximum temperature
  ■ Minimum temperature
  ■ Dominant weather condition (give reason on this)
○ Store the daily summaries in a database or persistent storage for further analysis.

2. Alerting Thresholds:

○ Define user-configurable thresholds for temperature or specific weather conditions (e.g., alert if temperature exceeds 35 degrees Celsius for two consecutive updates).
○ Continuously track the latest weather data and compare it with the thresholds.
○ If a threshold is breached, trigger an alert for the current weather conditions. Alerts could be displayed on the console or sent through an email notification system (implementation details left open-ended).

3. Implement visualizations:

○ To display daily weather summaries, historical trends, and triggered alerts.

Test Cases:

1. System Setup:

○ Verify system starts successfully and connects to the OpenWeatherMap API using a valid API key.

2. Data Retrieval:

○ Simulate API calls at configurable intervals.
○ Ensure the system retrieves weather data for the specified location and parses the response correctly.

3. Temperature Conversion:

○ Test conversion of temperature values from Kelvin to Celsius (or Fahrenheit) based on user preference.

4. Daily Weather Summary:

○ Simulate a sequence of weather updates for several days.
○ Verify that daily summaries are calculated correctly, including average, maximum, minimum temperatures,and dominant weather condition.

5. Alerting Thresholds:

○ Define and configure user thresholds for temperature or weather conditions.
○ Simulate weather data exceeding or breaching the thresholds.
○ Verify that alerts are triggered only when a threshold is violated.

Bonus:

- Extend the system to support additional weather parameters from the OpenWeatherMap API (e.g., humidity, wind speed) and incorporate them into rollups/aggregates.

- Explore functionalities like weather forecasts retrieval and generating summaries based on predicted conditions.

EXPLANATION

In this , part the datas are taken with the API that was taken from the Openweathermap so in this we are calculating the weather rate on every 5 minutes to ensure the value in the defined region.

I’m trying to satisfy all the contents given in the pdf.

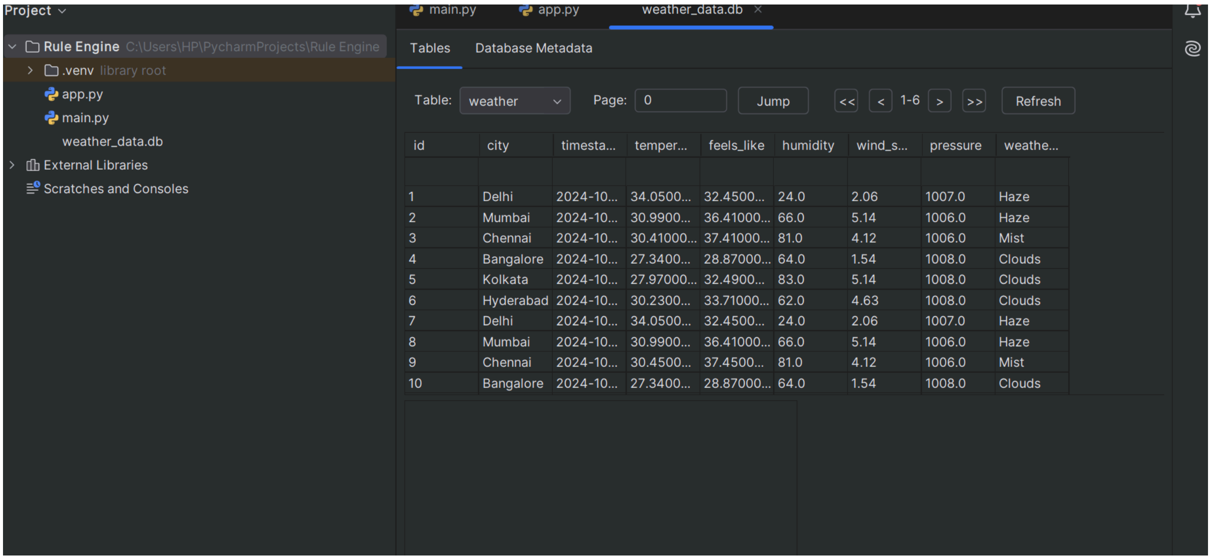
Here,the datas are going to their process to accumulate the region.

CODING

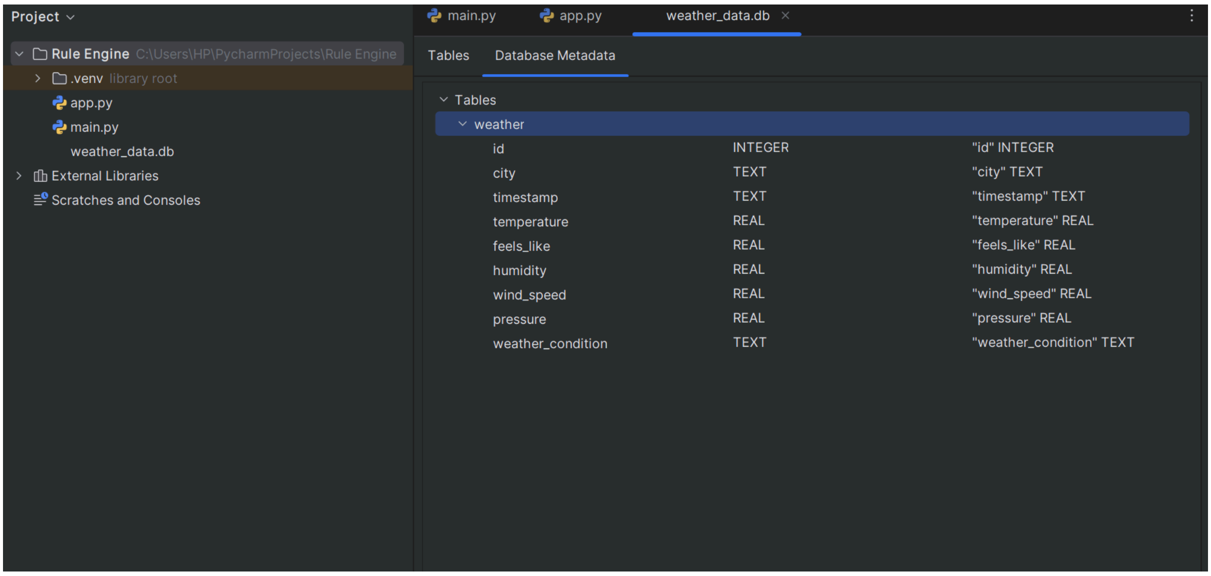
```
import requests
import time
import pandas as pd
import sqlite3
import plotly.graph_objects as go
from datetime import datetime
from collections import defaultdict

API_KEY = '879ef38780a6620d00008138626ad0be'  # OpenWeatherMap API key
CITIES = {
    "Delhi": 1273294,
    "Mumbai": 1275339,
    "Chennai": 1264527,
    "Bangalore": 1277333,
    "Kolkata": 1275004,
    "Hyderabad": 1269843
}
INTERVAL = 300  # 5 minutes interval for data fetch

weather_data = defaultdict(list)
```

```
# SQLite database setup
conn = sqlite3.connect('weather_data.db')
cursor = conn.cursor()

# Create a table for weather data if not exists
cursor.execute('''CREATE TABLE IF NOT EXISTS weather (
    id INTEGER PRIMARY KEY,
    city TEXT,
    timestamp TEXT,
    temperature  REAL,
    feels_like REAL,
    humidity REAL,
    wind_speed REAL,
    pressure REAL,
    weather_condition TEXT
)''')
conn.commit()


def kelvin_to_celsius(kelvin_temp):
    """Convert temperature from Kelvin to Celsius."""
    return kelvin_temp - 273.15


def fetch_weather(city_id):
    """Fetch current weather data from OpenWeatherMap for a given city."""
    url =
f"http://api.openweathermap.org/data/2.5/weather?id={city_id}&appid={API_KEY}"
    response = requests.get(url)
    return response.json()


def process_weather_data(data, city):
    """Process and store real-time weather data."""
    main_data = data['main']
    weather_condition = data['weather'][0]['main']
    temp_celsius = kelvin_to_celsius(main_data['temp'])
    feels_like_celsius = kelvin_to_celsius(main_data['feels_like'])
    humidity = main_data['humidity']
    wind_speed = data['wind']['speed']
    pressure = main_data['pressure']

    timestamp = datetime.utcfromtimestamp(data['dt']).strftime('%Y-%m-%d %H:%M:%S')
    # Append data to storage
    weather_data[city].append({
        "timestamp": timestamp,
        "temperature": temp_celsius,
        "feels_like": feels_like_celsius,
        "humidity": humidity,
        "wind_speed": wind_speed,
        "pressure": pressure,
        "weather_condition": weather_condition,
    })

    cursor.execute('''INSERT INTO weather (city, timestamp, temperature, feels_like,
humidity, wind_speed, pressure, weather_condition)
                      VALUES (?, ?, ?, ?, ?, ?, ?, ?)''',
                   (city, timestamp, temp_celsius, feels_like_celsius, humidity, wind_speed,
pressure,
                    weather_condition))
    conn.commit()


def aggregate_daily(city):
    """Calculate daily weather summary aggregates."""
    df = pd.DataFrame(weather_data[city])
    if len(df) > 0:
        df['timestamp'] = pd.to_datetime(df['timestamp'])
        df = df.set_index('timestamp')

        daily_summary = {
            'avg_temp': df['temperature'].mean(),
            'max_temp': df['temperature'].max(),
            'min_temp': df['temperature'].min(),
            'dominant_condition': df['weather_condition'].mode()[0]
        }
        return daily_summary
    return None


def check_threshold(city, temp, threshold=35):
    """Check if temperature exceeds the threshold and trigger an alert."""
    if temp > threshold:
        print(f"Alert: Temperature in {city} exceeded {threshold}°C!")


def plot_weather(city):
    """Enhanced real-time colorful visualization with additional parameters."""
    df = pd.DataFrame(weather_data[city])
    if len(df) > 0:
        fig = go.Figure()

        fig.add_trace(go.Scatter(x=df['timestamp'], y=df['temperature'], mode='lines+markers',
name='Temperature (°C)',
                                 line=dict(color='firebrick', width=2)))

        fig.add_trace(go.Scatter(x=df['timestamp'], y=df['feels_like'], mode='lines+markers',
name='Feels Like (°C)',
                                 line=dict(color='royalblue', width=2, dash='dash')))

        fig.add_trace(go.Bar(x=df['timestamp'], y=df['humidity'], name='Humidity (%)',
marker_color='lightblue'))

        fig.add_trace(go.Scatter(x=df['timestamp'], y=df['wind_speed'], mode='lines',
name='Wind Speed (m/s)',
                                 line=dict(color='green', width=2)))

        fig.add_trace(go.Scatter(x=df['timestamp'], y=df['pressure'], mode='lines',
name='Pressure (hPa)',
                                 line=dict(color='purple', width=2)))

        fig.update_layout(
            title=f"Real-Time Weather Data for {city}",
            xaxis_title="Time",
            yaxis_title="Value",
            template="plotly_dark",
            plot_bgcolor='rgba(0,0,0,0)',
            legend_title="Weather Metrics",
            font=dict(size=14)
        )

        # Show the plot
        fig.show()


def display_aggregates(city):
    """Display daily aggregate weather summary."""
    summary = aggregate_daily(city)
    if summary:
        print(f"Daily Weather Summary for {city}:")
        print(f"Avg Temp: {summary['avg_temp']:.2f}°C")
        print(f"Max Temp: {summary['max_temp']:.2f}°C")
        print(f"Min Temp: {summary['min_temp']:.2f}°C")
        print(f"Dominant Condition: {summary['dominant_condition']}")


def run_weather_monitoring():
    while True:
        for city, city_id in CITIES.items():
            try:
                # Fetch real-time weather data
                data = fetch_weather(city_id)
                process_weather_data(data, city)

                temp = kelvin_to_celsius(data['main']['temp'])
                check_threshold(city, temp, threshold=35)

                plot_weather(city)

                display_aggregates(city)

            except Exception as e:
                print(f"Error fetching data for {city}: {e}")

        # Sleep for the defined interval (e.g., 5 minutes)
        time.sleep(INTERVAL)


# Start the system
run_weather_monitoring()

from flask import Flask, request, jsonify


# Node class to represent conditions or operators in the tree (AST)
class Node:
    def __init__(self, node_type, left=None, right=None, value=None):
        self.node_type = node_type  # 'operator' (AND/OR) or 'operand' (condition like "age >
30")
        self.left = left  # Left child node (for AND/OR operators)
        self.right = right  # Right child node
        self.value = value  # Condition (e.g., "age > 30") or operator (e.g., "AND")


# Function to create the AST from a rule string
```

```
def create_rule(rule_string):
    # Split by AND (you can extend this to handle complex rules with OR, etc.)
    if "AND" in rule_string:
        conditions = rule_string.split("AND")
        left_condition = conditions[0].strip()
        right_condition = conditions[1].strip()

        # Create operand nodes for the conditions
        left_node = Node(node_type="operand", value=left_condition)
        right_node = Node(node_type="operand", value=right_condition)

        # Create a root node for the AND operator
        return Node(node_type="operator", value="AND", left=left_node, right=right_node)
    elif "OR" in rule_string:
        conditions = rule_string.split("OR")
        left_condition = conditions[0].strip()
        right_condition = conditions[1].strip()

        # Create operand nodes for the conditions
        left_node = Node(node_type="operand", value=left_condition)
        right_node = Node(node_type="operand", value=right_condition)

        # Create a root node for the OR operator
        return Node(node_type="operator", value="OR", left=left_node, right=right_node)


# Function to evaluate the AST against user data
def evaluate_rule(node, user_data):
    if node.node_type == "operand":
        # Extract the condition and split it into parts
        attribute, operator, threshold = node.value.split()
        user_value = user_data.get(attribute)

        # Simple evaluation logic (extend as needed)
        if operator == ">":
            return user_value > int(threshold)
        elif operator == "<":
            return user_value < int(threshold)
        elif operator == "==":
            return user_value == threshold

    elif node.node_type == "operator":
        # Recursively evaluate left and right child nodes
        left_result = evaluate_rule(node.left, user_data)
        right_result = evaluate_rule(node.right, user_data)

        # Return result based on operator type (AND/OR)
        if node.value == "AND":
            return left_result and right_result
        elif node.value == "OR":
            return left_result or right_result


# Initialize the Flask application
app = Flask(__name__)


# API endpoint to create a rule and return the AST (for testing purposes)
@app.route('/create_rule', methods=['POST'])
def create_rule_api():
    rule_string = request.json.get('rule')
    rule_ast = create_rule(rule_string)  # Parse rule string into AST
    return jsonify({"ast": str(rule_ast)})


# API endpoint to evaluate the rule against user data
@app.route('/evaluate_rule', methods=['POST'])
def evaluate_rule_api():
    rule_string = request.json.get('rule')  # Rule string (e.g., "age > 30 AND salary > 50000")
    user_data = request.json.get('data')  # User data (e.g., {"age": 35, "salary": 60000})

    rule_ast = create_rule(rule_string)  # Convert rule string into AST
    result = evaluate_rule(rule_ast, user_data)  # Evaluate the AST against user data

    return jsonify({"result": result})


# Start the Flask application
if __name__ == "__main__":
    app.run(debug=True)
```



id	city	timesta...	temper...	feels_like	humidity	wind_s...	pressure	weathe...
1	Delhi	2024-10...	34.0500...	32.4500...	24.0	2.06	1007.0	Haze
2	Mumbai	2024-10...	30.9900...	36.41000...	66.0	5.14	1006.0	Haze
3	Chennai	2024-10...	30.41000...	37.41000...	81.0	4.12	1006.0	Mist
4	Bangalore	2024-10...	27.3400...	28.87000...	64.0	1.54	1008.0	Clouds
5	Kolkata	2024-10...	27.97000...	32.4900...	83.0	5.14	1008.0	Clouds
6	Hyderabad	2024-10...	30.2300...	33.71000...	62.0	4.63	1008.0	Clouds
7	Delhi	2024-10...	34.0500...	32.4500...	24.0	2.06	1007.0	Haze
8	Mumbai	2024-10...	30.9900...	36.41000...	66.0	5.14	1006.0	Haze
9	Chennai	2024-10...	30.4500...	37.4500...	81.0	4.12	1006.0	Mist
10	Bangalore	2024-10...	27.3400...	28.87000...	64.0	1.54	1008.0	Clouds

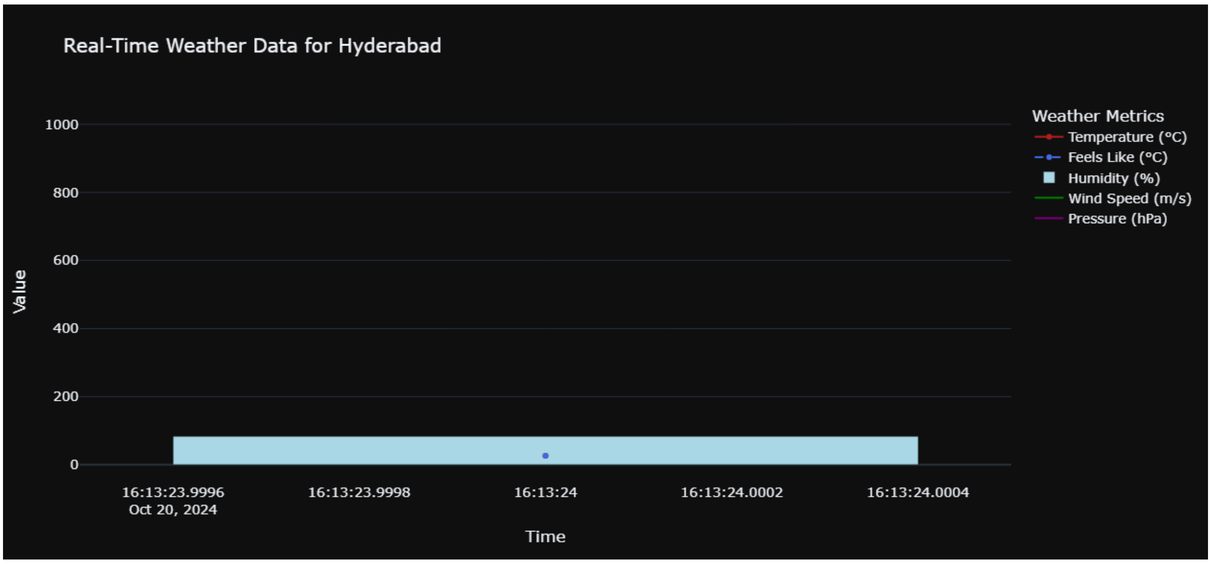


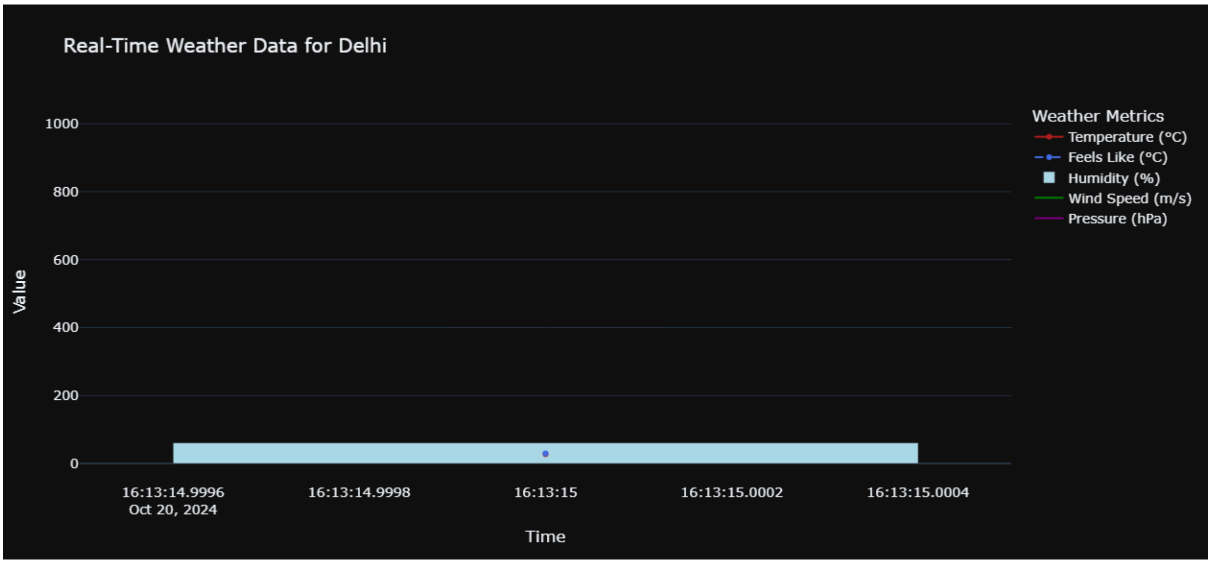
Tables

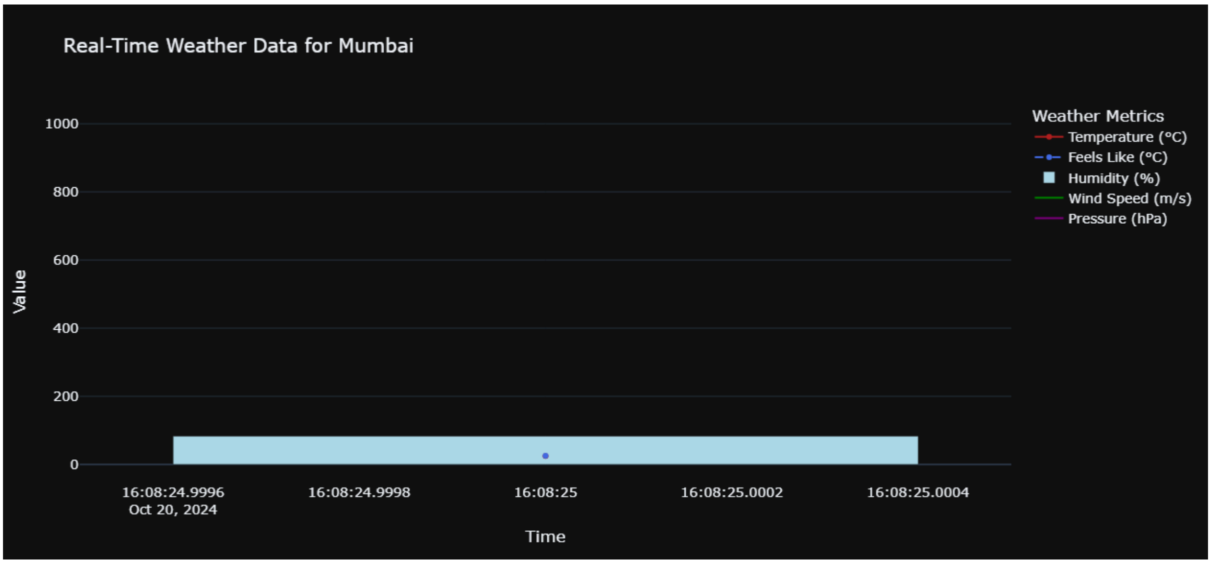
weather

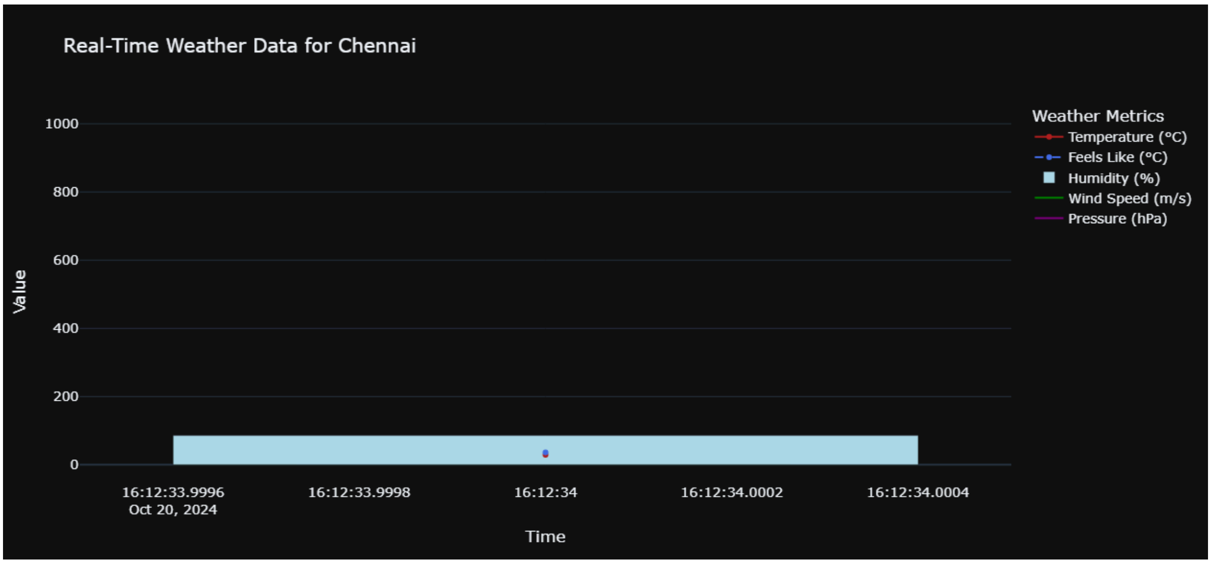
id	INTEGER	"id" INTEGER
city	TEXT	"city" TEXT
timestamp	TEXT	"timestamp" TEXT
temperature	REAL	"temperature" REAL
feels_like	REAL	"feels_like" REAL
humidity	REAL	"humidity" REAL
wind_speed	REAL	"wind_speed" REAL
pressure	REAL	"pressure" REAL
weather_condition	TEXT	"weather_condition" TEXT

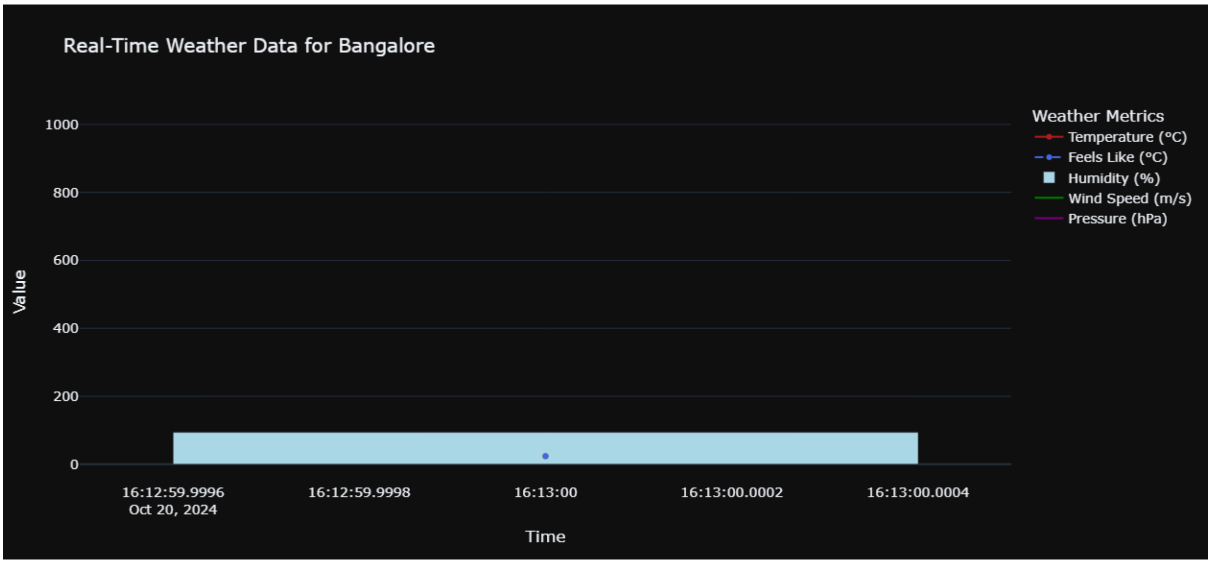
OUTPUT

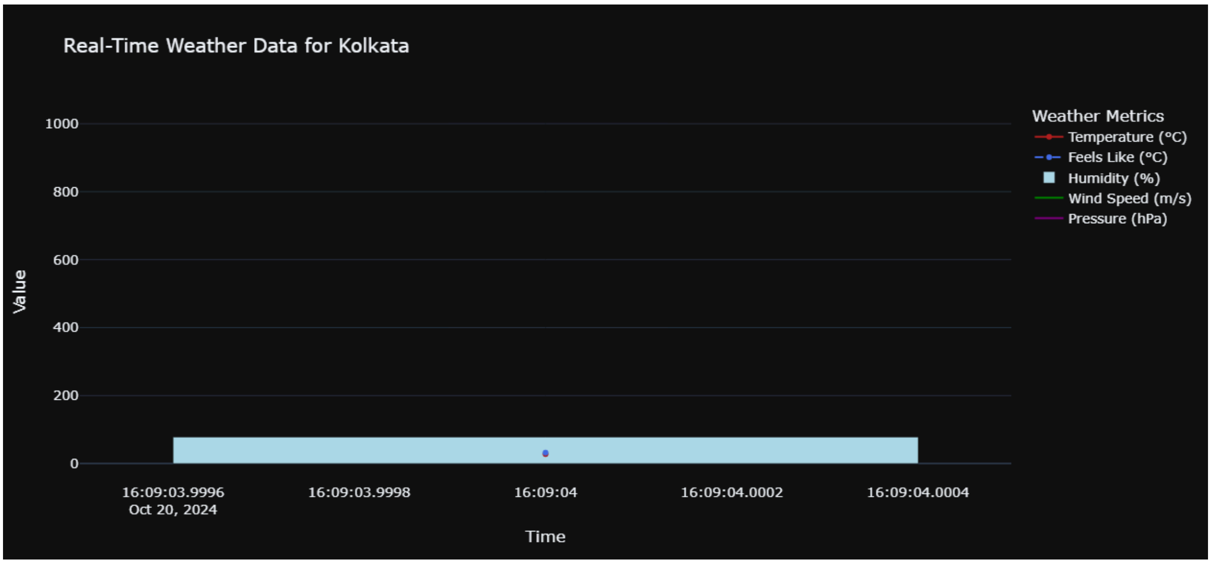


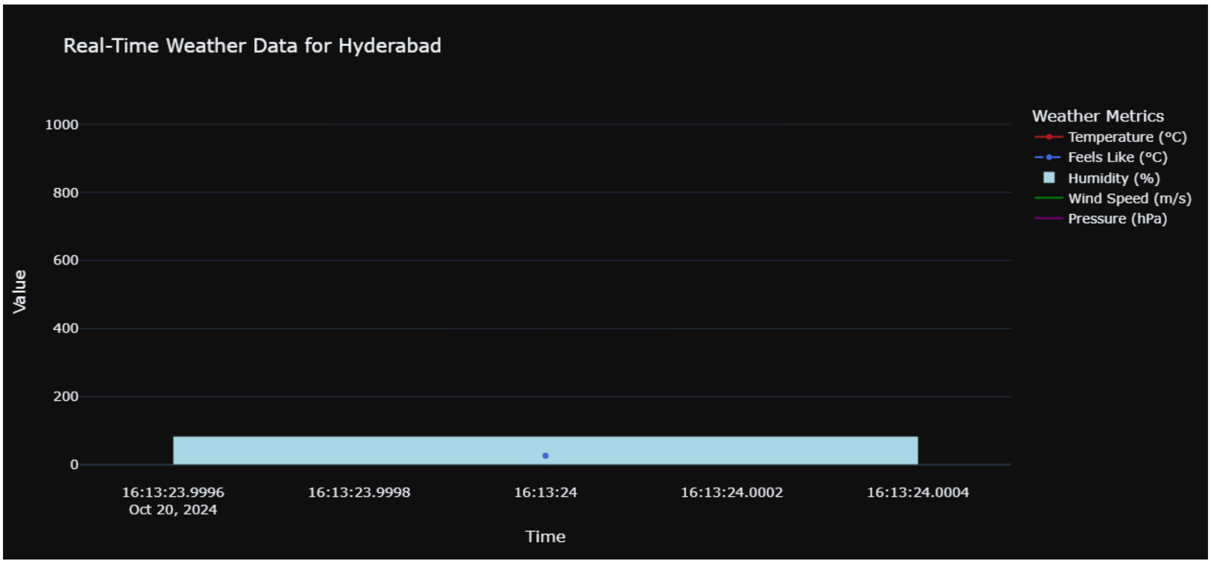

Real-Time Weather Data for Delhi
Value
1000
800
600
400
200
0
16:13:14.9996
Oct 20, 2024
16:13:14.9998
16:13:15
16:13:15.0002
16:13:15.0004
Time
Weather Metrics
Temperature (°C)
Feels Like (°C)
Humidity (%)
Wind Speed (m/s)
Pressure (hPa)


Real-Time Weather Data for Mumbai
Value
1000
800
600
400
200
0
16:08:24.9996
Oct 20, 2024
16:08:24.9998
16:08:25
16:08:25.0002
16:08:25.0004
Time
Weather Metrics
Temperature (°C)
Feels Like (°C)
Humidity (%)
Wind Speed (m/s)
Pressure (hPa)


Real-Time Weather Data for Chennai
Value
1000
800
600
400
200
0
16:12:33.9996
Oct 20, 2024
16:12:33.9998
16:12:34
16:12:34.0002
16:12:34.0004
Time
Weather Metrics
Temperature (°C)
Feels Like (°C)
Humidity (%)
Wind Speed (m/s)
Pressure (hPa)


Real-Time Weather Data for Bangalore
Value
1000
800
600
400
200
0
16:12:59.9996
Oct 20, 2024
16:12:59.9998
16:13:00
16:13:00.0002
16:13:00.0004
Time
Weather Metrics
Temperature (°C)
Feels Like (°C)
Humidity (%)
Wind Speed (m/s)
Pressure (hPa)


Real-Time Weather Data for Kolkata
Value
1000
800
600
400
200
0
16:09:03.9996
Oct 20, 2024
16:09:03.9998
16:09:04
16:09:04.0002
16:09:04.0004
Time
Weather Metrics
Temperature (°C)
Feels Like (°C)
Humidity (%)
Wind Speed (m/s)
Pressure (hPa)


Real-Time Weather Data for Hyderabad
Value
1000
800
600
400
200
0
16:13:23.9996
Oct 20, 2024
16:13:23.9998
16:13:24
16:13:24.0002
16:13:24.0004
Time
Weather Metrics
Temperature (°C)
Feels Like (°C)
Humidity (%)
Wind Speed (m/s)
Pressure (hPa)

```
"C:\Users\HP\PycharmProjects\Rule Engine\.venv\Scripts\python.exe"
"C:\Users\HP\PycharmProjects\Rule Engine\main.py"
C:\Users\HP\PycharmProjects\Rule Engine\main.py:63: DeprecationWarning:
datetime.datetime.utcfromtimestamp() is deprecated and scheduled for removal in a future
version. Use timezone-aware objects to represent datetimes in UTC:
datetime.datetime.fromtimestamp(timestamp, datetime.UTC).
  timestamp = datetime.utcfromtimestamp(data['dt']).strftime('%Y-%m-%d %H:%M:%S')
Daily Weather Summary for Delhi:
Avg Temp: 32.05°C
Max Temp: 32.05°C
Min Temp: 32.05°C
Dominant Condition: Haze
C:\Users\HP\PycharmProjects\Rule Engine\main.py:63: DeprecationWarning:

datetime.datetime.utcfromtimestamp() is deprecated and scheduled for removal in a future
version. Use timezone-aware objects to represent datetimes in UTC:
datetime.datetime.fromtimestamp(timestamp, datetime.UTC).

Daily Weather Summary for Mumbai:
Avg Temp: 29.99°C
Max Temp: 29.99°C
Min Temp: 29.99°C
Dominant Condition: Smoke
C:\Users\HP\PycharmProjects\Rule Engine\main.py:63: DeprecationWarning:

datetime.datetime.utcfromtimestamp() is deprecated and scheduled for removal in a future
version. Use timezone-aware objects to represent datetimes in UTC:
datetime.datetime.fromtimestamp(timestamp, datetime.UTC).

Daily Weather Summary for Chennai:
Avg Temp: 32.62°C
Max Temp: 32.62°C
Min Temp: 32.62°C
Dominant Condition: Haze
C:\Users\HP\PycharmProjects\Rule Engine\main.py:63: DeprecationWarning:

datetime.datetime.utcfromtimestamp() is deprecated and scheduled for removal in a future
version. Use timezone-aware objects to represent datetimes in UTC:
datetime.datetime.fromtimestamp(timestamp, datetime.UTC).

Daily Weather Summary for Bangalore:
Avg Temp: 26.99°C
Max Temp: 26.99°C
```

```
Min Temp: 26.99°C
Dominant Condition: Rain
C:\Users\HP\PycharmProjects\Rule Engine\main.py:63: DeprecationWarning:

datetime.datetime.utcfromtimestamp() is deprecated and scheduled for removal in a future
version. Use timezone-aware objects to represent datetimes in UTC:
datetime.datetime.fromtimestamp(timestamp, datetime.UTC).

Daily Weather Summary for Kolkata:
Avg Temp: 29.97°C
Max Temp: 29.97°C
Min Temp: 29.97°C
Dominant Condition: Smoke
C:\Users\HP\PycharmProjects\Rule Engine\main.py:63: DeprecationWarning:

datetime.datetime.utcfromtimestamp() is deprecated and scheduled for removal in a future
version. Use timezone-aware objects to represent datetimes in UTC:
datetime.datetime.fromtimestamp(timestamp, datetime.UTC).

Daily Weather Summary for Hyderabad:
Avg Temp: 30.73°C
Max Temp: 30.73°C
Min Temp: 30.73°C
Dominant Condition: Clouds


Process finished with exit code -1073741510 (0xC000013A: interrupted by Ctrl+C)
```